9.1 Grouping data: struct

Sometimes two data items are really aspects of the same data. For example, time might be recorded in hours and minutes, as in 4 hours and 23 minutes. Or a point on a plot might be recorded as x = 5, y = 7. Storing such data in separate variables, such as runTimeHours and runTimeMinutes, is not as clear as grouping that data into a single variable, like runTime, which might have subitems runTime.hourValue and runTime.minuteValue.

PARTICIPATION ACTIVITY 9.1.1: Naturally grouped data.

1) Select the pair forming part of a person's height (in U.S. units)
   - Feet and inches
   - Inches and salary
   - Pounds and ounces

2) Select the group of items indicating the change provided to a person who pays for a meal.
   - Ounce, gill, pint, quart, and gallon
   - Mile, furlong, yard, feet, and inches
   - Dollars, quarters, dimes, nickels, and pennies

The ***struct*** construct defines a new type, which can be used to declare a variable with subitems. The following animation illustrates.

PARTICIPATION ACTIVITY 9.1.2: A struct enables creating a variable with data members.

Animation content:

Code snippet is as follows:

```
typedef struct TimeHrMin_struct {
   int hourValue;
   int minuteValue;
} TimeHrMin;
```

```
...

   TimeHrMin runTime1;
   TimeHrMin runTime2;
   TimeHrMin runTime3;

   runTime1.hourValue = 5;
   runTime1.minuteValue = 46;
   runTime3.hourValue = runTime1.hourValue;

Final memory contents is as follows:
96 (runTime1's hourValue): 5
97 (runTime1's hourValue): 46
98 (runTime2's hourValue): ?
99 (runTime2's hourValue): ?
100 (runTime3's hourValue): 5
101 (runTime3's hourValue): ?
102: empty
```


Animation captions:

1. The struct construct just declares new type; no memory is allocated.
2. Variable definitions allocate memory for each object's member.
3. Accesses refer to an object member's memory location.

The programmer uses a struct to define and use a new type as follows.

Construct 9.1.1: Defining and using a new struct type.

```
typedef struct StructTypeName_struct {
   type item1;
   type item2;
   ...
   type itemN;
} StructTypeName;

...
StructTypeName myVar;

myVar.item1 = ...
```

The above uses a common combination of a typedef definition with a struct definition. A ***typedef*** defines a new type name for an existing type. This material uses that combination exclusively and does not discuss typedef definition separately.

The `struct StructTypeName_struct { ... }` part defines a new struct type named `struct StructTypeName_struct`. The typedef part defines a new type name named `StructTypeName` that is synonymous with `struct StructTypeName_struct`.

A programmer can use StructTypeName to declare a variable of that struct type as in the statement `StructTypeName myVar;`.

Each `type` may be any type like int or char. Each struct subitem is called a ***data member***. For a declared variable, each struct data member can be accessed using ".", known as a ***member access*** operator, sometimes called ***dot notation***.

Assigning a variable of a struct type to another such variable automatically assigns each corresponding data member, as shown below.

PARTICIPATION ACTIVITY 9.1.3: Assigning a struct type.

Animation content:

```
Code snippet is as follows:
typedef struct TimeHrMin_struct {
   int hourValue;
   int minuteValue;
} TimeHrMin;

...

   TimeHrMin runTime1;
   TimeHrMin runTime2;
   TimeHrMin runTime3;

   runTime1.hourValue = 5;
   runTime1.minuteValue = 46;
   runTime2 = runTime1;

Final memory contents is as follows:
96 (runTime1's hourValue): 5
97 (runTime1's hourValue): 46
98 (runTime2's hourValue): ?
99 (runTime2's hourValue): ?
100 (runTime3's hourValue): 5
```

```
101 (runTime3's hourValue): ?
102: empty
```

Animation captions:

1. Assigning a variable of a struct type to another such variable automatically assigns each corresponding data member.

Forgetting to include the semicolon at the end of a struct definition will generate cryptic compilation errors:

Figure 9.1.1: Less-than-helpful error message when forgetting the semicolon at the end of a struct definition.

```
gcc -Wall testfile.c
testfile.c:6: error: two or more data types in declaration specifiers
testfile.c:6: warning: return type of 'main' is not 'int'
testfile.c: In function 'main':
testfile.c:7: error: incompatible types in return
testfile.c:8: warning: control reaches end of non-void function
```

Try 9.1.1: Internet search for clues of error message cause.

Do an Internet search by copying and pasting the following (from the second line of the above figure):

error: two or more data types in declaration specifiers

Then, read over the first 3 search results, particularly focusing on the reply messages to find clues to the error message's cause.

PARTICIPATION ACTIVITY 9.1.4: The struct construct.

1) A struct definition for CartesianPoint has subitems int x and int y. How many int locations in memory does the struct definition allocate?

Check Show answer

2) If struct definition CartesianPoint has subitems int x and int y, how many total int locations in memory are allocated for these variable declarations?


```
int myNum;
CartesianPoint myPoint1;
CartesianPoint myPoint2;
```

Check Show answer

3) Given time1 is of type TimeHrMin defined earlier. What is the value of variable min after the following statements?

```
time1.hourValue   = 5;
time1.minuteValue = 4;
min = (60 * time1.hourValue) +
time1.minuteValue;
```

Check Show answer

4) Write a statement to assign 12 to the hourValue data member of TimeHrMin variable time1.

Check Show answer

5) Write a statement that assigns the value of the hourValue data member of time1 into the hourValue data member of time2.

Check Show answer

6) Write a single statement that assigns the values of all data members of time1 to the corresponding data members of time2.

Check Show answer

7) Declare a variable person1 of type Person, where Person is already defined as a struct type.

Check Show answer

CHALLENGE ACTIVITY 9.1.1: Enter the output using struct.

347282.2052980.qx3zqy7

Start

Type the program's output

```
#include <stdio.h>

typedef struct Height_struct {
   int feet;
   int inches;
} Height;

int main() {
   Height annHeight;

   annHeight.feet = 4;
   annHeight.inches = 10;

   printf("Ann: %dft %d\n", annHeight.feet, annHeight.inches);

   return 0;
}
```

```
Ann: 4ft
```


1 2 3 4

Check Next

CHALLENGE ACTIVITY 9.1.2: Defining a struct.

Define a struct named PatientData that contains two integer data members named heightInches and weightPounds. Sample output for the given program with inputs 63 115:

```
Patient data: 63 in, 115 lbs
```


347282.2052980.qx3zqy7

```c
#include <stdio.h>

/* Your solution goes here  */

int main(void) {
   PatientData lunaLovegood;

   scanf("%d", &lunaLovegood.heightInches);
   scanf("%d", &lunaLovegood.weightPounds);

   printf("Patient data: %d in, %d lbs\n", lunaLovegood.heightInches, lunaLovegood.

   return 0;
}
```

Run

View your last submission

CHALLENGE ACTIVITY 9.1.3: Accessing a struct's data members.

Write a statement to print the data members of InventoryTag. End with newline. Ex: if itemID is 314 and quantityRemaining is 500, print:

```
Inventory ID: 314, Qty: 500
```

347282.2052980.qx3zqy7

```c
#include <stdio.h>

typedef struct InventoryTag_struct {
   int itemID;
   int quantityRemaining;
} InventoryTag;

int main(void) {
   InventoryTag redSweater;

   scanf("%d", &redSweater.itemID);
   scanf("%d", &redSweater.quantityRemaining);

   /* Your solution goes here  */

   return 0;
}
```


Run

View your last submission

9.2 Pointers with structs

The malloc() function is commonly used with struct types to allocate the appropriate block of memory for a variable of a struct type.

Figure 9.2.1: Using malloc() with a struct type.

```
num1: 5
num2: 10
```

```
#include <stdio.h>
#include <stdlib.h>

typedef struct myItem_struct {
   int num1;
   int num2;
} myItem;

void myItem_PrintNums(myItem* itemPtr) {
   if (itemPtr == NULL) return;

   printf("num1: %d\n", itemPtr->num1);
   printf("num2: %d\n", itemPtr->num2);
}

int main(void) {
   myItem* myItemPtr1 = NULL;

   myItemPtr1 = (myItem*)malloc(sizeof(myItem));

   myItemPtr1->num1 = 5;
   (*myItemPtr1).num2 = 10;

   myItem_PrintNums(myItemPtr1);

   return 0;
}
```


Accessing a struct's member variables by first dereferencing a pointer, as in `(*myItemPtr1).num2 = 10;`, is so common that the language includes a second ***member access operator*** with the form:

Construct 9.2.1: Member access operator.

```
structPtr->memberName     // Equivalent to (*structPtr).memberName
```

The above program illustrates use of the member access operator: `myItemPtr1->num1 = 5;`.

PARTICIPATION ACTIVITY 9.2.1: The malloc, free, and -> operators.

Assuming the following is defined:

```
typedef struct Fruit_struct {
   // member variables
} Fruit;
```

1)

Declare a variable named orange as a pointer of type Fruit.

Check Show answer

2) Write a statement that allocates memory for the new variable orange that points to class Fruit.

Check Show answer

3) For the variable orange, write a statement that assigns a member variable named hasSeeds to 1. Use the -> operator.

Check Show answer

4) Write a statement that deallocates memory pointed to by variable orange, which is a pointer of type Fruit.

Check Show answer

5) Assuming a struct Fruit has two int data members and an int is 32 bits, what number does sizeof(Fruit) return?

Check Show answer

CHALLENGE ACTIVITY 9.2.1: Enter the output of points and structs.

347282.2052980.qx3zqy7

Start

Type the program's output

```
#include <stdio.h>
#include <stdlib.h>

typedef struct Home_struct {
   int numRooms;
   int numBaths;
} Home;

void Home_PrintHome(Home* itemPtr) {
   if (itemPtr == NULL) return;

   printf("%d rooms, %d baths\n", itemPtr->numRooms, itemPtr->numBaths);
}

int main(void) {
   Home* myHome = NULL;

   myHome = (Home*)malloc(sizeof(Home));

   myHome->numRooms = 6;
   myHome->numBaths = 2;

   Home_PrintHome(myHome);

   return 0;
}
```


6 room

1 2

Check Next

CHALLENGE ACTIVITY 9.2.2: Struct pointers.

Write two statements to assign numApples with 10 and numOranges with 3. Sample output for given program:

```
Apples: 10
Oranges: 3
```


347282.2052980.qx3zqy7

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
```

```
typedef struct bagContents_struct {
   int numApples;
   int numOranges;
} bagContents;

void bagContents_PrintBag(bagContents* itemPtr) {
   if (itemPtr == NULL) return;

   printf("Apples: %d\n", itemPtr->numApples);
   printf("Oranges: %d\n", itemPtr->numOranges);
}

int main(void) {
   bagContents* groceryPtr = NULL;

```


Run

View your last submission

9.3 Structs and functions

The struct construct's power is evident when used with functions. A struct can be used to return multiple values. Although ConvHrMin() has two output values, the struct type allows the function to return a single item, avoiding a less-clear approach using two pass by reference parameters.

PARTICIPATION ACTIVITY 9.3.1: Using a struct that is returned from a function; the struct's data members are copied upon return.

Animation content:

Code snippet is as follows:

```
#include
using namespace std;

struct TimeHrMin {
   int hourValue;
   int minuteValue;
};

TimeHrMin ConvHrMin(int totalTime) {
   TimeHrMin timeStruct;
```

```
   timeStr.hourValue  = totalTime / 60;
   timeStr.minuteValue = totalTime % 60;

   return timeStruct;
}

int main() {
   int inTime;
   TimeHrMin travelTime;

   cout << "Enter total minutes: ";
   cin >> inTime;

   travelTime = ConvHrMin(inTime);

   cout << "Equals: ";
   cout << travelTime.hourValue << " hrs ";
   cout << travelTime.minuteValue << " mins";

   return 0;
}

Final memory contents is as follows:
96 (main's inTime): 156
97 (main's travelTime hourValue): 2
98 (main's's travelTime hourValue): 36
99: empty
100 (ConvHrMin's totTime): 156
101 (ConvHrMin's timeStruct hourValue): 2
102 (ConvHrMin's timeStruct minuteValue): 36
```


Animation captions:

1. The program prompts a user to enter travel time in minutes, then calls the ConvHrMin function to convert travel time to hours and minutes.
2. Upon return, timeStruct's data members are copied to main's travelTime variable.
3. Returning a struct type allows the ConvHrMin function to return a single item instead of using two pass-by-reference parameters.

zyDE 9.3.1: Monetary change program.

Complete the program to compute monetary change, using the largest coins possible.

Load default template...

```
#include <stdio.h>

typedef struct MonetaryChange_struct
   int quarters;
   // FIXME: Finish data members
} MonetaryChange;

MonetaryChange ComputeChange(int cer
   MonetaryChange change;

   // FIXME: Finish function
   change.quarters = 0; // FIXME

   return change;
}

int main(void) {
   int userCents;
```

119

Run

PARTICIPATION ACTIVITY 9.3.2: Functions returning struct values.

1) Complete the function definition for a function ComputeLocation that returns a struct of type GPSPosition.

```
(double latitude, double
longitude) {

     ...

}
```

Check Show answer

2) Complete the function to return the calculated elapsed time, which gets stored in elapsedTime.

```
TimeEntry CalcElapsedTime(int
startSecs, int endSecs) {

     TimeEntry elapsedTime;

     ...

     elapsedTime.totalSecs =
endSecs - startSecs;

     elapsedTime.hours =
(endSecs - startSecs) / 3600;

     ...


                              ;

}
```

Check Show answer

Likewise, a variable of a struct type can be a function parameter. And just like other types, a pass by value parameter would copy the item, while a pass by reference parameter would not.

PARTICIPATION ACTIVITY 9.3.3: Functions with struct parameters.

1) Complete the function definition for a function CalcSpeed that returns a double value and has two struct type parameters startLoc and endLoc (in that order) of type GPSPosition.

```
double CalcSpeed(
                              
) {
    ...
}
```

Check Show answer

2) Complete the following statement to calculate the speed between gpsPos1 and gpsPos2 by making a call to the CalcSpeed function.

```
double vehicleSpeed;
GPSPosition gpsPos1;
GPSPosition gpsPos2;
...
vehicleSpeed =
;
...
```

Check Show answer

CHALLENGE ACTIVITY 9.3.1: Enter the output of the struct and function.

347282.2052980.qx3zqy7

Start

Type the program's output

```
#include <stdio.h>;

typedef struct Home_struct {
   int numBathrooms;
   int numFloors;
   int numPeople;
   int numRooms;
} Home;

Home InitHome() {
   Home tempHome;

   tempHome.numBathrooms = 4;
   tempHome.numFloors = 3;
   tempHome.numPeople = 9;
   tempHome.numRooms = 6;

   return tempHome;
}

int main() {
   Home myHome = InitHome();

   printf("%d Floors\n", myHome.numFloors);
   printf("%d People\n", myHome.numPeople);

   return 0;
}
```

```
3 Floors
9 People
```


1 2 3

Check Next

CHALLENGE ACTIVITY 9.3.2: Structs and functions.

347282.2052980.qx3zqy7

Start

Write a statement that calls the function AddToStock with parameters notebookInfo and addStock. Assign notebookInfo with the returned value.


```
#include <stdio.h>

typedef struct ProductInfo_struct {
   char itemName[50];
   int itemQty;
} ProductInfo;

ProductInfo AddToStock(ProductInfo productToStock, int increaseValue) {
   productToStock.itemQty = productToStock.itemQty + increaseValue;

   return productToStock;
}

int main(void) {
   ProductInfo notebookInfo;
   int addStock;

   scanf("%s", notebookInfo.itemName);
   scanf("%d", &notebookInfo.itemQty);
   scanf("%d", &addStock);

   /* Your code goes here */

   printf("Name: %s, stock: %d\n", notebookInfo.itemName, notebookInfo.itemQty

   return 0;
}
```

1 2 3

Check Next

9.4 Structs and arrays

The power of structs becomes even more evident when used in conjunction with arrays. Consider a TV watching time program where a user can enter a country name, and the program outputs the daily TV watching hours average for a person in that country. One approach uses two same-sized arrays,

one to hold names, and the other to hold numbers corresponding to each name. Instead of those two arrays, a struct allows for declaration of just one array that stores items that each have name and number data members.

Figure 9.4.1: An array of struct items rather than two arrays of more basic types.


```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

const int MAX_COUNTRY_NAME_LENGTH = 50;

typedef struct CountryTvWatch_struct {
   char countryName[MAX_COUNTRY_NAME_LENGTH];
   int tvMinutes;
} CountryTvWatch;

int main(void) {
   // Source: www.statista.com, 2010
   const int NUM_COUNTRIES = 4;

   CountryTvWatch countryList[NUM_COUNTRIES];
   char countryToFind[MAX_COUNTRY_NAME_LENGTH];
   bool countryFound;
   int i;

   strcpy(countryList[0].countryName, "Brazil");
   countryList[0].tvMinutes = 222;
   strcpy(countryList[1].countryName, "India");
   countryList[1].tvMinutes = 119;
   strcpy(countryList[2].countryName, "U.K.");
   countryList[2].tvMinutes = 242;
   strcpy(countryList[3].countryName, "U.S.A.");
   countryList[3].tvMinutes = 283;

   printf("Enter country name: ");
   scanf("%s", countryToFind);

   countryFound = false;
   for (i = 0; i < NUM_COUNTRIES && !countryFound; ++i) { // Find
country's index
      if (strcmp(countryList[i].countryName, countryToFind) == 0) {
         countryFound = true;
         printf("People in %s watch\n", countryToFind);
         printf("%d minutes of TV daily.\n",
countryList[i].tvMinutes);
      }
   }
   if (!countryFound) {
      printf("Country not found, try again.\n");
   }

   return 0;
}
```

```
Enter country name:
U.S.A.
People in U.S.A. watch
283 minutes of TV daily.
...
Enter country name: UK
Country not found, try
again.
...
Enter country name: U.K.
People in U.K. watch
242 minutes of TV daily.
```

Note that the countryList variable is declared as `CountryTvWatch countryList[NUM_COUNTRIES]`, meaning an array of items of type CountryTvWatch. Thus, each array element will have memory allocated for the struct's two data members, countryName and tvMinutes.

The notation `countryList[i].countryName` is equivalent to `(countryList[i]).countryName`, because the member access operator is evaluated left-to-right (as are any equal-precedence operators). The left-to-right member access operator evaluation is well-known among programmers so parentheses are typically omitted.

PARTICIPATION ACTIVITY 9.4.1: Using structs with arrays.

1) Declare the array countryList of 5 CountryTvWatch elements

Check Show answer

2) Given an array countryList consisting of 5 CountryTvWatch struct elements, write a statement that assigns the value of the 0th element's tvMinutes data member to the variable countryMin.

Check Show answer

3) Given an array countryList consisting of 5 CountryTvWatch struct elements, write one statement that copies element 4's struct values to element 0's.

Check Show answer

zyDE 9.4.1: Modify the TV watch program.

Finish the PrintCountryNames() function to print all country names in the list.

Load default template...

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

const int MAX_COUNTRY_NAME_LENGTH

typedef struct CountryTvWatch_stru
   char countryName[50];
   int tvMinutes;
} CountryTvWatch;

void PrintCountryNames(CountryTvWa
{
   printf("FIXME: Finish PrintCoun
}

int main(void) {
```

USA

Run

CHALLENGE ACTIVITY 9.4.1: Enter the output of the array of structs.

347282.2052980.qx3zqy7

Start

Type the program's output

```
#include <stdio.h>
#include <string.h>

typedef struct Car_struct {
   char type[20];
   char color[20];
} Car;

int main(void) {
   Car carList[3];

   strcpy(carList[0].type, "sedan");
   strcpy(carList[0].color, "red");
   strcpy(carList[1].type, "SUV");
   strcpy(carList[1].color, "white");
   strcpy(carList[2].type, "truck");
   strcpy(carList[2].color, "orange");

   printf("%s %s\n", carList[2].color, carList[2].type);
   printf("%s %s\n", carList[0].color, carList[0].type);

   return 0;
}
```


1 2 3

Check Next

CHALLENGE ACTIVITY 9.4.2: Structs and arrays.

347282.2052980.qx3zqy7

Start

Declare an array named pizzasInStore that stores 3 items of type Pizza.

```
#include <stdio.h>
#include <string.h>

typedef struct Pizza_struct {
   char pizzaName[75];
   char ingredients[75];
} Pizza;

int main(void) {

   /* Your code goes here */

   strcpy(pizzasInStore[0].pizzaName, "Barbecue");
   strcpy(pizzasInStore[1].pizzaName, "Carbonara");
   strcpy(pizzasInStore[2].pizzaName, "Ham and Cheese");
   strcpy(pizzasInStore[0].ingredients, "Beef, chicken, bacon, barbecue sauce"
   strcpy(pizzasInStore[1].ingredients, "Mushrooms, onion, creamy sauce");
   strcpy(pizzasInStore[2].ingredients, "Ham, cheese, bacon");

   printf("%s: %s\n", pizzasInStore[0].pizzaName, pizzasInStore[0].ingredients
   printf("%s: %s\n", pizzasInStore[1].pizzaName, pizzasInStore[1].ingredients
   printf("%s: %s\n", pizzasInStore[2].pizzaName, pizzasInStore[2].ingredients

   return 0;
}
```

1 2 3

Check Next

9.5 Structs, arrays, and functions: A seat reservation example

A programmer commonly uses structs, arrays, and functions together. Consider a program that allows a reservations agent to reserve seats for people, useful for a theater, an airplane, etc. The below program defines a Seat struct whose data members are a person's first name, last name, and the amount paid for the seat. The program declares an array of 5 seats to represent the theater, airplane, etc., initializes all seats to being empty (indicated by a first name of "empty"), and then allows a user to enter commands to print all seats, reserve a seat, or quit.

Figure 9.5.1: A seat reservation system involving a struct, arrays, and functions.

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

typedef struct Seat_struct {
   char firstName[50];
   char lastName[50];
   int  amountPaid;
} Seat;

/*** Functions for Seat ***/

void SeatMakeEmpty(Seat* seat) {
   strcpy((*seat).firstName, "empty");
   strcpy((*seat).lastName,  "empty");
   (*seat).amountPaid = 0;
}

bool SeatIsEmpty(Seat seat) {
   return (strcmp(seat.firstName, "empty") == 0);
}

void SeatPrint(Seat seat) {
   printf("%s ", seat.firstName);
   printf("%s, ", seat.lastName);
   printf("Paid: %d\n", seat.amountPaid);
}
/*** End functions for Seat ***/

/*** Functions for array of Seat ***/
void SeatsMakeEmpty(Seat seats[], int numSeats) {
   int i;

   for (i = 0; i < numSeats; ++i) {
      SeatMakeEmpty(&seats[i]);
   }
}

void SeatsPrint(Seat seats[], int numSeats) {
   int i;

   for (i = 0; i < numSeats; ++i) {
      printf("%d: ", i);
      SeatPrint(seats[i]);
   }
}
/*** End functions for array of Seat ***/

int main(void) {
   const int NUM_SEATS = 5;
```

```
Enter command (p/r/q): p
0: empty empty, Paid: 0
1: empty empty, Paid: 0
2: empty empty, Paid: 0
3: empty empty, Paid: 0
4: empty empty, Paid: 0

Enter command (p/r/q): r
Enter seat num: 2
Enter first name: John
Enter last name: Smith
Enter amount paid: 500
Completed.

Enter command (p/r/q): p
0: empty empty, Paid: 0
1: empty empty, Paid: 0
2: John Smith, Paid: 500
3: empty empty, Paid: 0
4: empty empty, Paid: 0

Enter command (p/r/q): r
Enter seat num: 2
Seat not empty.

Enter command (p/r/q): r
Enter seat num: 3
Enter first name: Mary
Enter last name: Jones
Enter amount paid: 198
Completed.

Enter command (p/r/q): p
0: empty empty, Paid: 0
1: empty empty, Paid: 0
2: John Smith, Paid: 500
3: Mary Jones, Paid: 198
4: empty empty, Paid: 0

Enter command (p/r/q): q
Quitting.
```

```
   char userKey;
   int  seatNum;
   Seat allSeats[NUM_SEATS];
   Seat currSeat;

   userKey = '-';

   SeatsMakeEmpty(allSeats, NUM_SEATS);

   while (userKey != 'q') {
      printf("Enter command (p/r/q): ");
      scanf(" %c", &userKey);

      if (userKey == 'p') { // Print seats
         SeatsPrint(allSeats, NUM_SEATS);
         printf("\n");
      }
      else if (userKey == 'r') { // Reserve seat
         printf("Enter seat num: ");
         scanf("%d", &seatNum);

         if (!SeatIsEmpty(allSeats[seatNum])) {
            printf("Seat not empty.\n\n");
         }
         else {
            printf("Enter first name: ");
            scanf("%s", currSeat.firstName);
            printf("Enter last name: ");
            scanf("%s", currSeat.lastName);
            printf("Enter amount paid: ");
            scanf("%d", &currSeat.amountPaid);

            allSeats[seatNum] = currSeat;

            printf("Completed.\n\n");
         }
      }
      // FIXME: Add option to delete reservations
      else if (userKey == 'q') { // Quit
         printf("Quitting.\n");
      }
      else {
         printf("Invalid command.\n\n");
      }
   }

   return 0;
}
```


The programmer first defined several functions related to the Seat struct, such as checking if a seat is empty or printing a seat. The programmer then defined some functions related to an *array* of seat items. To distinguish, the programmer named the former starting with Seat and the latter starting with Seats.

The SeatMakeEmpty() function uses pass by pointer to update the information within an individual seat. Remember that to update a variable passed by pointer, the program must use the dereference operator * to access the value pointed to by the pointer. When the variable is a pointer to a structure, both the dereference operator and the member access operators must be used together. In this case, the member access operator has precedence, so parentheses are used to dereference the pointer first:

Construct 9.5.1: Dereferencing a pointer to a struct.

```
(*variableName).memberName
```


The programmer left a "FIXME" comment indicating that the program also requires the ability to delete a reservation. That functionality is straightforward to introduce, just requiring the user to enter a seat number and then making use of the existing SeatMakeEmpty() function.

Notice how main() is relatively clean, dealing mostly with the user commands, and then using functions to carry out the appropriate work. Actually, the "reserve seat" command could be improved; main() currently fills the reservation information (e.g., "Enter first name..."), but main() would be cleaner if it just called a function as `SeatFillReservationInfo(&currSeat)`.

The seat reservation program loses all its information when exited. An improvement is to save all reservation information in a file. Commands 's' and 'g' would save and get information to/from a file, respectively.

PARTICIPATION ACTIVITY 9.5.1: Seat reservation example with struct, array, and functions.

Refer to the above example.

1) The number of seats is 5.
   - True
   - False

2) SeatsMakeEmpty() has a loop that sets each seat in the seats array to have a first name of "empty".
   - True
   - False

3) SeatIsEmpty() checks if all the seats in the array are empty.
   - True
   - False

4) Deleting a reservation would reduce the array size from 5 down to 4.
   - True

False

zyDE 9.5.1: Introduce delete behavior to the reservation program.

Modify main() to allow the user to enter command 'd', followed by the user entering a se number. Call SeatMakeEmpty() to delete the seat.

Load default template...

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

typedef struct Seat_struct {
   char firstName[50];
   char lastName[50];
   int  amountPaid;
} Seat;

/*** Functions for Seat ***/

void SeatMakeEmpty(Seat* seat) {
   strcpy((*seat).firstName, "empty"
   strcpy((*seat).lastName,  "empty"
   (*seat).amountPaid = 0;
}
```

```
p
r 2 John Smith 500
p
```

Run

9.6 Vector ADT

Structs and pointers can be used to implement a computing concept known as an ***abstract data type (ADT)***, which is a data type whose creation and update are supported by specific well-defined operations. A key aspect of an ADT is that the internal implementation of the data and operations are hidden from the ADT user, a concept known as ***information hiding***, thus allowing the ADT user to be more productive by focusing on higher-level concepts, and also allowing the ADT developer to improve the internal implementation without requiring changes to programs using the ADT.

Programmers commonly refer to separating an ADT's *interface* from its *implementation*; the user of an ADT need only know the ADT's interface (functions declared within the ADT's header file) and not its implementation (function definitions and struct data members).

PARTICIPATION ACTIVITY 9.6.1: Abstract data types (ADT).

1) ADTs (Abstract data types) are meant to hide the values of variables from the user.
   - True
   - False

2) An ADT's interface is commonly separated from its implementation.
   - True
   - False

3) An ADT is a fixed data type, like an int or char.
   - True
   - False

A vector is an example of ADT that stores an ordered list of items (called elements) of a given data type, such as a vector of integers. Like an array, an individual element within the vector can be accessed using an index. However, unlike an array, a vector's size can be adjusted while a program is executing, an especially useful feature when the number of items that will be in the list is unknown at compile-time. To support such size adjustment, a vector ADT provides functions for common operations like creating the vector, inserting elements into the vector, removing elements from the vector, resizing the vector, and accessing elements at specific locations.

The following defines an ADT for a vector of integers — defining a new struct type named "vector" and the function prototypes for the vector's interface. These declarations are included in the header file for the ADT named "vector.h".

Vector ADT naming convention

For the vector ADT, this material uses a different convention for naming the functions. All functions supporting the vector ADT begin with "vector_", which is intended to indicate the functions are associated with the "vector" type. The following portion of the function corresponds to operation being performed on the vector ADT.

Figure 9.6.1: struct and function prototypes for vector ADT.

```
#ifndef VECTOR_H
#define VECTOR_H

// struct and typedef declaration for Vector ADT
typedef struct vector_struct {
   int* elements;
   unsigned int size;
} vector;

// interface for accessing Vector ADT

// Initialize vector
void vector_create(vector* v, unsigned int vectorSize);

// Destroy vector
void vector_destroy(vector* v);

// Resize the size of the vector
void vector_resize(vector* v, unsigned int vectorSize);

// Return pointer to element at specified index
int* vector_at(vector* v, unsigned int index);

// Insert new value at specified index
void vector_insert(vector* v, unsigned int index, int value);

// Insert new value at end of vector
void vector_push_back(vector* v, int value);

// Erase (remove) value at specified index
void vector_erase(vector* v, unsigned int index);

// Return number of elements within vector
int vector_size(vector* v);

#endif
```


To use the vector, a program must first include the following: `#include "vector.h"`. Then, a variable for a vector can be declared and used as shown in the below animation. The definition creates a vector variable named v. Data members within the vector struct are not initialized automatically. The vector_create() function is used to initialize the vector given the specified size of the vector. The function call `vector_create(&v, 3)` initializes the vector by allocating memory for three elements, each of type int, and initializes each of those elements to the value 0.

The function call `vector_at(&v, 0)` returns a pointer to the int element stored at location 0 of the vector v. A value can be written to this location by dereferencing the returned int pointer. For example, the statement `*vector_at(&v, 0) = 119;` assigns a value of 119 to the int element at location 0 of the vector v.

Notice that the first parameter for each of the vector's functions is a pointer to a vector ("vector*"). When calling each of these functions, this parameter will point to the specific vector on which the

corresponding operation will be performed. Thus, a program can consist of multiple vectors, using the same set of the vector functions.

PARTICIPATION ACTIVITY 9.6.2: Example use of vector ADT.

Animation captions:

1. A vector variable v is declared. The function call vector_create(&v, 3) allocates memory for three elements, each of type int, and initializes each of those elements to 0.
2. The function call vector_at(&v, index) returns a pointer to the element at the location index. A value can be written to this location by dereferencing the returned int pointer.
3. Similarly, a value can be read from this location by dereferencing the returned int pointer.

The following summarizes the functions for the vector ADT defined above:

Table 9.6.1: Vector ADT functions.

Function	Description	Example
`void vector_create(vector* v, unsigned int vectorSize)`	Initializes the vector pointed to by v with vectorSize number of elements. Each element with the vector is initialized to 0.	`vector_create(&v, 20)`
`void vector_destroy(vector* v)`	Destroys the vector by freeing all memory allocated within the vector.	`vector_destroy(&v)`
`void vector_resize(vector* v, unsigned int vectorSize)`	Resizes the vector with vectorSize number of elements. If the vector size increased, each new element within the vector is initialized to 0.	`vector_resize(&v, 25)`
`int* vector_at(vector* v, unsigned int index)`	Returns a pointer to the element at the location index.	`x = *vector_at(&v, 1)`
`unsigned int vector_size(vector* v)`	Returns the vector's size — i.e. the number of elements	`if (vector_size(&v) == 0)`

	within the vector.	
`void vector_push_back(vector* v, int value)`	Inserts the value x to a new element at the end of the vector, increasing the vector size by 1.	`// adds "47" onto the end of the vector` `vector_push_back(&v, 47);`

`void vector_insert(vector* v, unsigned int index, int value)`	Inserts the value x to the element indicated by position, making room by shifting over the elements at that position and higher, thus increasing the vector size by 1.	`// inserts "33" at index 2`
		`// Before: 18, 26, 47, 52`
		`// After: 18, 26, 33, 47, 52`
		`vector_insert(&v, 2, 33);`

`void vector_erase(vector* v, unsigned int index)`	Removes the element from the indicated position. All elements at higher positions are shifted over to fill the gap left by the removed element. Thus, the vector size decreases by 1.	`// erases element at index 0`
		`// Before: 18, 26, 47, 52`
		`// After: 26, 47, 52`
		`vector_erase(&v, 0);`

The definition of the vector's functions are implemented within a file named "vector.c" shown below. Notice from the vector's struct definition that the vector has a data member for a *dynamically allocated array* of integers and a data member for the size of the array. For ADTs, a programmer should not directly access the data members of the struct in order to adhere to the concept to information hiding.

Figure 9.6.2: Function definitions for vector ADT.

```
#include <stdio.h>
#include <stdlib.h>
#include "vector.h"

// Initialize vector with specified size
void vector_create(vector* v, unsigned int vectorSize) {
   int i;

   if (v == NULL) return;

   v->elements = (int*)malloc(vectorSize * sizeof(int));;
   v->size = vectorSize;
   for (i = 0; i < v->size; ++i) {
      v->elements[i] = 0;
   }
}
```

```
// Destroy vector
void vector_destroy(vector* v) {
   if (v == NULL) return;

   free(v->elements);
   v->elements = NULL;
   v->size = 0;
}

// Resize the size of the vector
void vector_resize(vector* v, unsigned int vectorSize) {
   int oldSize;
   int i;

   if (v == NULL) return;

   oldSize = v->size;
   v->elements = (int*)realloc(v->elements, vectorSize * sizeof(int));
   v->size = vectorSize;
   for (i = oldSize; i < v->size; ++i) {
      v->elements[i] = 0;
   }
}

// Return pointer to element at specified index
int* vector_at(vector* v, unsigned int index) {
   if (v == NULL || index >= v->size) return NULL;

   return &(v->elements[index]);
}

// Insert new value at specified index
void vector_insert(vector* v, unsigned int index, int value) {
   int i;

   if (v == NULL || index > v->size) return;

   vector_resize(v, v->size + 1);
   for (i = v->size - 1; i > index; --i) {
      v->elements[i] = v->elements[i-1];
   }
   v->elements[index] = value;
}

// Insert new value at end of vector
void vector_push_back(vector* v, int value) {
   vector_insert(v, v->size, value);
}

// Erase (remove) value at specified index
void vector_erase(vector* v, unsigned int index) {
   int i;

   if (v == NULL || index >= v->size) return;

   for (i = index; i < v->size - 1; ++i) {
      v->elements[i] = v->elements[i+1];
   }
   vector_resize(v, v->size - 1);
}

// Return number of elements within vector
int vector_size(vector* v) {
   if (v == NULL) return -1;

   return v->size;
}
```

A common error when using ADT such as a vector is passing an incorrect pointer to the vector's functions. For example, if a NULL pointer is passed to the vector_create() function, the expression `v->elements` would attempt to dereference a NULL pointer. As a NULL ptr refers to an invalid memory address, this type of error will cause a program to terminate in an error referred to as a ***segmentation fault***, ***access violation***, or ***bad access***. The actual error name depends on the computer system you are using.

To avoid this common error, each of the functions for the vector ADT first checks if the vector pointer is NULL before trying to dereference that pointer. If the pointer is NULL, the function will either return immediately or return a value indicating an error condition. For example, the function call `vector_size(NULL)` returns the value -1, indicating an error, as a vector cannot have a negative number of elements.

The following animation illustrates `vector_insert()` and `vector_erase()`.

PARTICIPATION ACTIVITY 9.6.3: Vector ADT functions.

Animation captions:

1. A variable vector v is declared and initialized.
2. The vector_erase( ) function removes an element from the indicated position. All elements at higher positions are shifted over and the vector size is decreased by one.
3. The vector_insert( ) function inserts a value in the indicated by position. All elements in a higher position are shifted over and the vector size is increased by one.
4. The vector_size( ) function returns the number of elements within the vector. The vector_destroy( ) function frees all memory allocated within the vector.

The following modifies an earlier number smoothing program to use the vector ADT rather than directly using an array. The benefit is that the program can support an arbitrary number of user-entered integer numbers.

Figure 9.6.3: Number smoothing program using vector.

```c
#include <stdio.h>
#include <stdlib.h>
#include "vector.h"

// Get number from user
void GetNums(vector* nums) {
   int numsSize;     // Vector size
   int i;            // Loop index

   printf("Enter number of integers to be entered: ");
   scanf("%d", &numsSize);

   vector_resize(nums, numsSize);

   for (i = 0; i < vector_size(nums); ++i) {
      printf("%d: ", i + 1);
      scanf("%d", vector_at(nums, i));
   }
}

// Smooths by setting element to average of itself and next
two elements
void FilterNums(vector* nums) {
   int i;

   for (i = 0; i < vector_size(nums) - 2; ++i) {
      *vector_at(nums, i) = (*vector_at(nums, i) +
                             *vector_at(nums, i + 1) +
                             *vector_at(nums, i + 2)) / 3;
   }

   *vector_at(nums, i) = (*vector_at(nums, i) +
                          *vector_at(nums, i + 1)) / 2;

   // Last element needs no averaging
}

// Print all elements within the vector
void PrintsNums(vector* nums) {
   int i;

   printf("Numbers: ");
   for (i = 0; i < vector_size(nums); ++i) {
      printf("%d ", *vector_at(nums, i));
   }
   printf("\n");
}

int main(void) {
   vector nums;

   vector_create(&nums, 0);

   GetNums(&nums);
   PrintsNums(&nums);
   FilterNums(&nums);
   PrintsNums(&nums);

   vector_destroy(&nums);

   return 0;
}
```

```
Enter number of integers to be
entered: 10
1: 10
2: 20
3: 30
4: 40
5: 50
6: 60
7: 70
8: 80
9: 90
10: 100
Numbers: 10 20 30 40 50 60 70 80
90 100
Numbers: 20 30 40 50 60 70 80 90
95 100
```

PARTICIPATION ACTIVITY 9.6.4: Vector declaration and use.

Write a single statement for each answer.

1) Declare a vector named vals.

Check Show answer

2) Initialize a vector vals to size 10 with all elements set to 0.

Check Show answer

3) Assign the value of the element held at index 8 of vector vals to an int variable x.

Check Show answer

4) Write 555 into element at index 2 of vector vals.

Check Show answer

5) Store 777 into the second element of vector vals.

Check Show answer

6) Append the value 37 to the vector vals.

Check Show answer

7) Set the int variable sz to the size of the vector vals.

Check Show answer

8) Erase element 0 of vector vals.

Check Show answer

CHALLENGE ACTIVITY 9.6.1: Modifying a vector.

Modify the existing vector's contents, by erasing 200, then inserting 100 and 102 in the shown locations. Use Vector ADT's erase() and insert() only. Sample output of below program:

```
100 101 102 103
```

347282.2052980.qx3zqy7

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  // struct and typedef declaration for Vector ADT
5  typedef struct vector_struct {
6     int* elements;
7     unsigned int size;
8  } vector;
9
10 // Initialize vector with specified size
11 void vector_create(vector* v, unsigned int vectorSize) {
12    int i;
13
14    if (v == NULL) return;
15
16    v->elements = (int*)malloc(vectorSize * sizeof(int));
17    v->size = vectorSize;
18    for (i = 0; i < v->size; ++i) {
```

Run

View your last submission

9.7 C example: Employee list using vector ADT

zyDE 9.7.1: Managing an employee list using a vector ADT.

The following program allows a user to add to and list entries from a vector ADT, which maintains a list of employees.

1. Run the program, and provide input to add three employees' names and related da
   Then use the list option to display the list.
2. Modify the program to implement the deleteEntry function.
3. Run the program again and add, list, delete, and list again various entries.

Current file: **main.c** Load default templ

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <ctype.h>
4  #include "vector_employee.h"
5
6  // Add an employee
7  void AddEmployee(vector *employees) {
8     Employee theEmployee;
9
10    printf("\nEnter the name to add: \n");
11    fgets(theEmployee.name, 50, stdin);
12    theEmployee.name[strlen(theEmployee.name) - 1] = '\0'; // Remove trail
13
14    printf("Enter %s's department: \n", theEmployee.name);
15    fgets(theEmployee.department, 50, stdin);
16    theEmployee.department[strlen(theEmployee.department) - 1] = '\0'; //
17
18    printf("Enter %s's title: \n", theEmployee.name);
```



```
a
Rajeev Gupta
Sales
```

Run

Below is a solution to the above problem.

zyDE 9.7.2: Managing an employee list using a vector ADT (solution).

Current file: **main.c** Load default templ

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <ctype.h>
4  #include "vector_employee.h"
5
6  // Add an employee
7  void AddEmployee(vector *employees) {
8     Employee theEmployee;
9
10    printf("\nEnter the name to add: \n");
11    fgets(theEmployee.name, 50, stdin);
12    theEmployee.name[strlen(theEmployee.name) - 1] = '\0'; // Remove trail
13
14    printf("Enter %s's department: \n", theEmployee.name);
15    fgets(theEmployee.department, 50, stdin);
16    theEmployee.department[strlen(theEmployee.department) - 1] = '\0'; //
17
18    printf("Enter %s's title: \n", theEmployee.name);
```

```
a
Rajeev Gupta
Sales
```

Run

9.8 Why pointers: A list example

To further elaborate on the need for pointers, this section describes another of many situations where pointers are useful.

The vector ADT (or arrays) stores a list of items in contiguous memory locations, which enables immediate access to any element i of vector v by using `vector_at(&v, i)`. Recall that inserting an item within a vector requires making room by shifting higher-indexed items. Similarly, erasing an item requires shifting higher-indexed items to fill the gap. Each shift of an item from one element to another element requires a few processor instructions. This issue exposes the ***vector insert/erase performance problem***. For vectors with thousands of elements, a single call to insert() or erase() can require thousands of instructions, so if a program does many inserts or erases on large vectors, the program may run very slowly. The following animation illustrates shifting during an insert.

PARTICIPATION ACTIVITY 9.8.1: Vector insert performance problem.

Animation captions:

1. Inserting an item at a specific location in a vector requires making room for the item by shifting higher-indexed items.

The following program can be used to demonstrate the issue. The user inputs a vector size vectorSize, and a number numOps of elements to insert. The program then carries out several tasks, namely it resizes the vector to size vectorSIze, writes an arbitrary value to all vectorSIze elements, does numOps push_backs, numOps inserts, and numOps erases.

Figure 9.8.1: Program illustrating how slow vector inserts and erases can be.

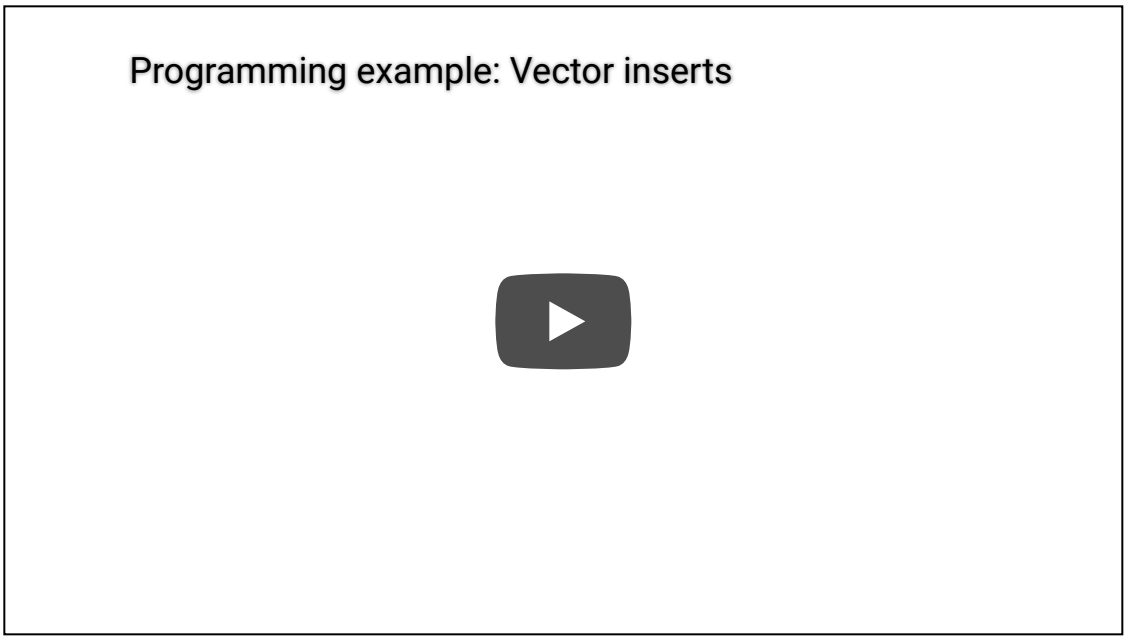
```c
#include <stdio.h>
#include <stdlib.h>
#include "vector.h"

int main(void) {
   vector tempValues;  // Dummy vector to demo vector ops
   int vectorSize;     // User defined vector size
   int numOps;         // User defined number of inserts
   int i;              // Loop index

   vector_create(&tempValues, 0);

   printf("Enter initial vector size: ");
   scanf("%d", &vectorSize);

   printf("Enter number of inserts: ");
   scanf("%d", &numOps);

   printf("  Resizing vector...");
   fflush(stdout);
   vector_resize(&tempValues, vectorSize);

   printf("done.\n");
   printf("  Writing to each element...");
   fflush(stdout);

   for (i = 0; i < vectorSize; ++i) {
      *vector_at(&tempValues, i) = 777; // Any value
   }

   printf("done.\n");
   printf("   Doing %d inserts at end...", numOps);
   fflush(stdout);

   for (i = 0; i < numOps; ++i) {
      vector_insert(&tempValues, vector_size(&tempValues), 888); // Any value
   }

   printf("done.\n");
   printf("   Doing %d inserts at beginning...", numOps);
   fflush(stdout);

   for (i = 0; i < numOps; ++i) {
      vector_insert(&tempValues, 0, 444);
   }

   printf("done.\n");
   printf("   Doing %d removes...", numOps);
   fflush(stdout);


   for (i = 0; i < numOps; ++i) {
      vector_erase(&tempValues, 0);
   }

   printf("done.\n");

   return 0;
}
```


©zyBooks 12/22/21 12:35 1026490
Jacob Adams
UACS100Fall2021

```
Enter initial vector size: 100000
Enter number of inserts: 40000
  Resizing vector...done.                      (fast)
  Writing to each element...done.              (fast)
   Doing 40000 inserts at end...done.          (fast)
   Doing 40000 inserts at beginning...done.    (SLOW)
   Doing 40000 removes...done.                 (SLOW)
```

The video shows the program running for different vectorSize and numOps values; notice that for large vectorSize and numOps, the resize, writes, and numOps push_backs all run quickly, but the numOps inserts and numOps erases take a noticeably long time. The `fflush(stdout);` forces any characters written to stdout to be displayed on the screen before doing each task, lest the characters be held in the buffer until a task completes.

Video 9.8.1: Vector inserts.



The vector_push_backs() are fast because they do not involve any shifting of elements, whereas each vector_insert() requires 500,000 elements to be shifted — one at a time. 7,500 inserts thus requires 3,750,000,000 (over 3 billion) shifts.

One way to make inserts or erases faster is to use a different approach for storing a list of items. The approach does not use contiguous memory locations. Instead, each item contains a "pointer" to the next item's location in memory, as well as, the data being stored. Thus, inserting a new item B between existing items A and C just requires changing A to point to B's memory location, and B to point to C's location, as shown in the following animation.

PARTICIPATION ACTIVITY 9.8.2: A list avoids the shifting problem.

Animation captions:

1. List's first two items initially: (A, C, ...). Item A points to the next item at location 88. Item C points to next item at location 113 (not shown).
2. To insert new item B after item A, the new item B is first added to memory at location 90.
3. Item A is updated to point to location 90. Item B is set to point to location 88. New list is (A, B, C, ...). No shifting of items after C was required.

The animation begins with a list having some number of items, with the first two items being A and C. The first item has data A, and an address 88 pointing to the next item's location in memory, which just happens to be adjacent to the first item's location. That second item has data C, and an address 113 pointing to the next item (not shown). The animation shows a new item being created at memory location 90, having data B. To keep the list in sorted order, item B should go between A and C in the list. So item A's next pointer is changed to point to B's location of 90, and B's next pointer is set to point to 88.

A ***linked list*** is a list wherein each item contains not just data but also a pointer — a *link* — to the next item in the list. Comparing vectors and linked lists:

- *Vector*: Stores items in contiguous memory locations. Supports quick access to i'th element via `vector_at(&v, i)`, but may be slow for inserts or deletes on large lists due to necessary shifting of elements.
- *Linked list*: Stores each item anywhere in memory, with each item pointing to the next item in the list. Supports fast inserts or deletes, but access to i'th element may be slow as the list must be traversed from the first item to the i'th item. Also uses more memory due to storing a link for each item.

PARTICIPATION ACTIVITY 9.8.3: Vector performance.

1) Appending a new item to the end of a 1000 element vector requires how many elements to be shifted?

Check Show answer

2) Inserting a new item at the beginning of a 1000 element vector requires how many elements to be shifted?

Check Show answer

9.9 A first linked list

A common use of pointers is to create a list of items such that an item can be efficiently inserted somewhere in the middle of the list, without the shifting of later items as required for a vector. The following program illustrates how such a list can be created. A struct is defined to represent each list item, known as a ***list node***. A node is comprised of the data to be stored in each list item, in this case just one int, and a pointer to the next node in the list. A special node named head is created to represent the front of the list, after which regular items can be inserted.

Figure 9.9.1: A basic example to introduce linked lists.

```
#include <stdio.h>
#include <stdlib.h>

typedef struct IntNode_struct {
   int dataVal;
   struct IntNode_struct* nextNodePtr;
} IntNode;

// Constructor
void IntNode_Create
(IntNode* thisNode, int dataInit, IntNode* nextLoc) {
   thisNode->dataVal = dataInit;
   thisNode->nextNodePtr = nextLoc;
}

/* Insert newNode after node.
 Before: thisNode -- next
 After:  thisNode -- newNode -- next
 */
void IntNode_InsertAfter
(IntNode* thisNode, IntNode* newNode) {
   IntNode* tmpNext = NULL;

   tmpNext = thisNode->nextNodePtr; // Remember next
   thisNode->nextNodePtr = newNode; // this -- new -- ?
   newNode->nextNodePtr = tmpNext;  // this -- new -- next
}

// Print dataVal
void IntNode_PrintNodeData(IntNode* thisNode) {
   printf("%d\n", thisNode->dataVal);
}

// Grab location pointed by nextNodePtr
IntNode* IntNode_GetNext(IntNode* thisNode) {
   return thisNode->nextNodePtr;
}

int main(void) {
   IntNode* headObj  = NULL; // Create IntNode objects
   IntNode* nodeObj1 = NULL;
   IntNode* nodeObj2 = NULL;
   IntNode* nodeObj3 = NULL;
```

```
-1
555
777
999
```

```
   IntNode* currObj  = NULL;

   // Front of nodes list
   headObj = (IntNode*)malloc(sizeof(IntNode));
   IntNode_Create(headObj, -1, NULL);

   // Insert nodes
   nodeObj1 = (IntNode*)malloc(sizeof(IntNode));
   IntNode_Create(nodeObj1, 555, NULL);
   IntNode_InsertAfter(headObj, nodeObj1);

   nodeObj2 = (IntNode*)malloc(sizeof(IntNode));
   IntNode_Create(nodeObj2, 999, NULL);
   IntNode_InsertAfter(nodeObj1, nodeObj2);

   nodeObj3 = (IntNode*)malloc(sizeof(IntNode));
   IntNode_Create(nodeObj3, 777, NULL);
   IntNode_InsertAfter(nodeObj1, nodeObj3);

   // Print linked list
   currObj = headObj;
   while (currObj != NULL) {
      IntNode_PrintNodeData(currObj);
      currObj = IntNode_GetNext(currObj);
   }

   return 0;
}
```


PARTICIPATION ACTIVITY 9.9.1: Inserting nodes into a basic linked list.

Animation captions:

1. The headObj pointer points to a special node that represents the front of the list. When the list is first created, no list items exists, so the head node's nextNodePtr pointer is null.
2. To insert a node in the list, the new node nodeObj1 is first created with the value 555.
3. To insert the new node, tmpNext is pointed to the head node's next node, the head node's nextNodePtr is pointed to the new node, and the new node's nextNodePtr is pointed to tmpNext.
4. A second node nodeObj2 with the value 999 is inserted at the end of the list, and a third node nodeObj3 with the value 777 is created.
5. To insert nodeObj3 after nodeObj1, tmpNext is pointed to the nodeObj1's next node, the nodeObj1's nextNodePtr is pointed to the nodeObj3, and nodeObj3's nextNodePtr is pointed to tmpNext.

The most interesting part of the above program is the InsertAfter() function, which inserts a new node after a given node already in the list. The above animation illustrates.

PARTICIPATION ACTIVITY 9.9.2: A first linked list.

Some questions refer to the above linked list code and animation.

1) A linked list has what key advantage over a sequential storage approach like an array or vector?

   - An item can be inserted somewhere in the middle of the list without having to shift all subsequent items.
   - Uses less memory overall.
   - Can store items other than int variables.

2) What is the purpose of a list's head node?

   - Stores the first item in the list.
   - Provides a pointer to the first item's node in the list, if such an item exists.
   - Stores all the data of the list.

3) After the above list is done having items inserted, at what memory address is the last list item's node located?

   - 80
   - 82
   - 84
   - 86

4) After the above list has items inserted as above, if a fourth item was inserted at the front of the list, what would happen to the location of node1?

   - Changes from 84 to 86.
   - Changes from 84 to 82.
   - Stays at 84.

In contrast to the above program that declares one variable for each item allocated by the malloc function, a program commonly declares just one or a few variables to manage a large number of

items allocated using the malloc function. The following example replaces the above main() function, showing how just two pointer variables, currObj and lastObj, can manage 20 allocated items in the list.

Figure 9.9.2: Managing many new items using just a few pointer variables.


```
-1
1481765933
1085377743
1270216262
1191391529
812669700
553475508
445349752
1344887256
730417256
1812158119
147699711
880268351
1889772843
686078705
2105754108
182546393
1949118330
220137366
1979932169
1089957932
```

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct IntNode_struct {
   int dataVal;
   struct IntNode_struct* nextNodePtr;
} IntNode;

// Constructor
void IntNode_Create
(IntNode* thisNode, int dataInit, IntNode* nextLoc) {
   thisNode->dataVal = dataInit;
   thisNode->nextNodePtr = nextLoc;
}

/* Insert newNode after node.
 Before: thisNode -- next
 After:  thisNode -- newNode -- next
 */
void IntNode_InsertAfter
(IntNode* thisNode, IntNode* newNode) {
   IntNode* tmpNext = NULL;

   tmpNext = thisNode->nextNodePtr; // Remember next
   thisNode->nextNodePtr = newNode;// this -- new -- ?
   newNode->nextNodePtr = tmpNext; // this -- new -- next
}

// Print dataVal
void IntNode_PrintNodeData(IntNode* thisNode) {
   printf("%d\n", thisNode->dataVal);
}

// Grab location pointed by nextNodePtr
IntNode* IntNode_GetNext(IntNode* thisNode) {
   return thisNode->nextNodePtr;
}

int main(void) {
   IntNode* headObj  = NULL; // Create intNode objects
   IntNode* currObj  = NULL;
   IntNode* lastObj  = NULL;
   int i;                    // Loop index

   headObj = (IntNode*)malloc(sizeof(IntNode)); // Front of nodes list
   IntNode_Create(headObj, -1, NULL);
   lastObj = headObj;

   for (i = 0; i < 20; ++i) {                      // Append 20 rand nums
      currObj = (IntNode*)malloc(sizeof(IntNode));
      IntNode_Create(currObj, rand(), NULL);

      IntNode_InsertAfter(lastObj, currObj);    // Append curr
      lastObj = currObj;                        // Curr is the new last item
   }


   currObj = headObj;                           // Print the list

   while (currObj != NULL) {
      IntNode_PrintNodeData(currObj);
      currObj = IntNode_GetNext(currObj);
   }

   return 0;
}
```

zyDE 9.9.1: Managing a linked list.

Finish the program so that it finds and prints the smallest value in the linked list.

Load default template... Run

```
1
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  typedef struct IntNode_struct {
6     int dataVal;
7     struct IntNode_struct* nextNodePt
8  } IntNode;
9
10 // Constructor
11 void IntNode_Create
12 (IntNode* thisNode, int dataInit, Ir
13    thisNode->dataVal = dataInit;
14    thisNode->nextNodePtr = nextLoc;
15 }
16
17 /* Insert newNode after node.
18
```

Normally, a linked list would be implemented as an ADT using a set interface functions and struct type declaration, such as IntList. Data members of that struct might include a pointer to the list head, the list size, and a pointer to the list tail (the last node in the list). Supporting functions might include InsertAfter (insert a new node after the given node), PushBack (insert a new node after the last node), PushFront (insert a new node at the front of the list, just after the head), DeleteNode (deletes the node from the list), etc.

CHALLENGE ACTIVITY 9.9.1: Enter the output of the program using Linked List.

347282.2052980.qx3zqy7

Start

Type the program's output

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct PlaylistSong_struct {
   char name[50];
   struct PlaylistSong_struct* nextPlaylistSongPtr;
} PlaylistSong;

void PlaylistSong_Create(PlaylistSong* thisNode, char* name, PlaylistSong* nextLoc) {
   strcpy(thisNode->name, name);
   thisNode->nextPlaylistSongPtr = nextLoc;
}

void PlaylistSong_InsertAfter(PlaylistSong* thisNode, PlaylistSong* newNode) {
   PlaylistSong* tmpNext = NULL;

   tmpNext = thisNode->nextPlaylistSongPtr;
   thisNode->nextPlaylistSongPtr = newNode;
   newNode->nextPlaylistSongPtr = tmpNext;
}

void PlaylistSong_PrintNodeData(PlaylistSong* thisNode) {
   printf("%s\n", thisNode->name);
}

PlaylistSong* PlaylistSong_GetNext(PlaylistSong* thisNode) {
   return thisNode->nextPlaylistSongPtr;
}

int main(void) {
   PlaylistSong* headObj = NULL;
   PlaylistSong* firstSong = NULL;
   PlaylistSong* secondSong = NULL;
   PlaylistSong* thirdSong = NULL;
   PlaylistSong* currObj = NULL;

   headObj = (PlaylistSong*)malloc(sizeof(PlaylistSong));
   PlaylistSong_Create(headObj, "head", NULL);

   firstSong = (PlaylistSong*)malloc(sizeof(PlaylistSong));
   PlaylistSong_Create(firstSong, "Egmont", NULL);
   PlaylistSong_InsertAfter(headObj, firstSong);

   secondSong = (PlaylistSong*)malloc(sizeof(PlaylistSong));
   PlaylistSong_Create(secondSong, "Cavatina", NULL);
   PlaylistSong_InsertAfter(firstSong, secondSong);

   thirdSong = (PlaylistSong*)malloc(sizeof(PlaylistSong));
   PlaylistSong_Create(thirdSong, "Adagio", NULL);
   PlaylistSong_InsertAfter(secondSong, thirdSong);

   currObj = headObj;
   while (currObj != NULL) {
      PlaylistSong_PrintNodeData(currObj);
      currObj = PlaylistSong_GetNext(currObj);
   }

   return 0;
}
```


©zyBooks 12/22/21 12:35 1026490
Jacob Adams
UACS100Fall2021

1 2

Check Next

CHALLENGE ACTIVITY 9.9.2: Linked list negative values counting.

Assign negativeCntr with the number of negative values in the linked list, including the list head.

347282.2052980.qx3zqy7

```
#include <stdio.h>
#include <stdlib.h>

typedef struct IntNode_struct {
   int dataVal;
   struct IntNode_struct* nextNodePtr;
} IntNode;

// Constructor
void IntNode_Create(IntNode* thisNode, int dataInit, IntNode* nextLoc) {
   thisNode->dataVal = dataInit;
   thisNode->nextNodePtr = nextLoc;
}

/* Insert newNode after node.
 Before: thisNode -- next
 After: thisNode -- newNode -- next
*/
```

Run

View your last submission

9.10 LAB: Warm up: Contacts

This section has been set as optional by your instructor.

You will be building a linked list. Make sure to keep track of both the head and tail nodes.

(1) Create three files to submit.

- ContactNode.h - Struct definition, including the data members and related function declarations
- ContactNode.c - Related function definitions
- main.c - main() function

(2) Build the ContactNode struct per the following specifications:

- Data members
  - char contactName[50]
  - char contactPhoneNum[50]
  - struct ContactNode* nextNodePtr
- Related functions
  - void CreateContactNode(ContactNode* thisNode, char nameInit[], char phoneNumInit[], ContactNode* nextLoc) (2 pt)
  - void InsertContactAfter(ContactNode* thisNode, ContactNode* newNode) (2 pts)
    - Insert a new node after node
  - ContactNode* GetNextContact() (1 pt)
    - Return location pointed by nextNodePtr
  - void PrintContactNode()

Ex. of PrintContactNode() output:

```
Name: Roxanne Hughes
Phone number: 443-555-2864
```

(3) In main(), prompt the user for three contacts and output the user's input. Create three ContactNodes and use the nodes to build a linked list. (2 pts)

Ex:

```
Person 1
Enter name:
Roxanne Hughes
Enter phone number:
443-555-2864
You entered: Roxanne Hughes, 443-555-2864

Person 2
Enter name:
Juan Alberto Jr.
Enter phone number:
410-555-9385
You entered: Juan Alberto Jr., 410-555-9385

Person 3
Enter name:
Rachel Phillips
```

```
Enter phone number:
310-555-6610
You entered: Rachel Phillips, 310-555-6610
```

(4) Output the linked list. (2 pts)

Ex:


```
CONTACT LIST
Name: Roxanne Hughes
Phone number: 443-555-2864

Name: Juan Alberto Jr.
Phone number: 410-555-9385

Name: Rachel Phillips
Phone number: 310-555-6610
```


LAB ACTIVITY | 9.10.1: LAB: Warm up: Contacts | 0 / 9

Current file: **main.c** Load default template...

```
1  #include<stdio.h>
2  #include<stdlib.h>
3  #include<string.h>
4
5  #include "ContactNode.h"
6
7  int main(void) {
8
9     /* Type code here */
10
11    return 0;
12 }
```


Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first

box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

9.11 LAB*: Program: Playlist

This section has been set as optional by your instructor.

You will be building a linked list. Make sure to keep track of both the head and tail nodes.

(1) Create three files to submit.

- PlaylistNode.h - Struct definition and related function declarations
- PlaylistNode.c - Related function definitions
- main.c - main() function

Build the PlaylistNode class per the following specifications. Note: Some functions can initially be function stubs (empty functions), to be completed in later steps.

- Private data members
  - char uniqueID[50]
  - char songName[50]
  - char artistName[50]
  - int songLength

    - PlaylistNode* nextNodePtr
- Related functions
    - CreatePlaylistNode() (1 pt)
    - InsertPlaylistNodeAfter() (1 pt)
        - Insert a new node after node
    - SetNextPlaylistNode() (1 pt)
        - Set a new node to come after node
    - GetNextPlaylistNode()
        - Return location pointed by nextNodePtr
    - PrintPlaylistNode()

Ex. of PrintPlaylistNode output:

```
Unique ID: S123
Song Name: Peg
Artist Name: Steely Dan
Song Length (in seconds): 237
```

(2) In main(), prompt the user for the title of the playlist. (1 pt)

Ex:

```
Enter playlist's title:
JAMZ
```

(3) Implement the PrintMenu() function. PrintMenu() takes the playlist title as a parameter and outputs a menu of options to manipulate the playlist. (1 pt)

Ex:

```
JAMZ PLAYLIST MENU
a - Add song
r - Remove song
c - Change position of song
s - Output songs by specific artist
t - Output total time of playlist (in seconds)
o - Output full playlist
q - Quit
```


(4) Implement the ExecuteMenu() function that takes 3 parameters: a character representing the user's choice, a playlist title, and the pointer to the head node of a playlist. ExecuteMenu() performs

the menu options (described below) according to the user's choice, and returns the pointer to the head node of the playlist.(1 pt)

(5) In main(), call PrintMenu() and prompt for the user's choice of menu options. Each option is represented by a single character.

If an invalid character is entered, continue to prompt for a valid choice. When a valid option is entered, execute the option by calling ExecuteMenu() and overwrite the pointer to the head node of the playlist with the returned pointer. Then, print the menu, and prompt for a new option. Continue until the user enters 'q'. *Hint: Implement Quit before implementing other options.* (1 pt)

Ex:

```
JAMZ PLAYLIST MENU
a - Add song
r - Remove song
c - Change position of song
s - Output songs by specific artist
t - Output total time of playlist (in seconds)
o - Output full playlist
q - Quit

Choose an option:
```

(6) Implement "Output full playlist" menu option in ExecuteMenu(). If the list is empty, output: **`Playlist is empty`** (3 pts)

Ex:

```
JAMZ - OUTPUT FULL PLAYLIST
1.
Unique ID: SD123
Song Name: Peg
Artist Name: Steely Dan
Song Length (in seconds): 237

2.
Unique ID: JJ234
Song Name: All For You
Artist Name: Janet Jackson
Song Length (in seconds): 391

3.
```

```
Unique ID: J345
Song Name: Canned Heat
Artist Name: Jamiroquai
Song Length (in seconds): 330

4.
Unique ID: JJ456
Song Name: Black Eagle
Artist Name: Janet Jackson
Song Length (in seconds): 197

5.
Unique ID: SD567
Song Name: I Got The News
Artist Name: Steely Dan
Song Length (in seconds): 306
```


(7) Implement the "Add song" menu option in ExecuteMenu(). New additions are added to the end of the list. (2 pts)

Ex:

```
ADD SONG
Enter song's unique ID:
SD123
Enter song's name:
Peg
Enter artist's name:
Steely Dan
Enter song's length (in seconds):
237
```

(8) Implement the "Remove song" menu option in ExecuteMenu(). Prompt the user for the unique ID of the song to be removed.(4 pts)

Ex:

```
REMOVE SONG
Enter song's unique ID:
JJ234
"All For You" removed.
```

(9) Implement the "Change position of song" menu option in ExecuteMenu(). Prompt the user for the current position of the song and the desired new position. Valid new positions are 1 - *n* (the number of nodes). If the user enters a new position that is less than 1, move the node to the position 1 (the head). If the user enters a new position greater than *n*, move the node to position *n* (the tail). 6 cases will be tested:

- Moving the head node (1 pt)
- Moving the tail node (1 pt)
- Moving a node to the head (1 pt)
- Moving a node to the tail (1 pt)
- Moving a node up the list (1 pt)
- Moving a node down the list (1 pt)

Ex:

```
CHANGE POSITION OF SONG
Enter song's current position:
3
Enter new position for song:
2
"Canned Heat" moved to position 2
```

(10) Implement the "Output songs by specific artist" menu option in ExecuteMenu(). Prompt the user for the artist's name, and output the node's information, starting with the node's current position. (2 pt)

Ex:

```
OUTPUT SONGS BY SPECIFIC ARTIST
Enter artist's name:
Janet Jackson

2.
Unique ID: JJ234
Song Name: All For You
Artist Name: Janet Jackson
Song Length (in seconds): 391

4.
Unique ID: JJ456
Song Name: Black Eagle
```

```
Artist Name: Janet Jackson
Song Length (in seconds): 197
```

(11) Implement the "Output total time of playlist" menu option in ExecuteMenu(). Output the sum of the time of the playlist's songs (in seconds). (2 pts)

Ex:


```
OUTPUT TOTAL TIME OF PLAYLIST (IN SECONDS)
Total time: 1461 seconds
```


LAB ACTIVITY | 9.11.1: LAB*: Program: Playlist | 0 / 26

Current file: **PlaylistNode.c** Load default template...

```
1 /* Type your code here */
```

Develop mode **Submit mode**

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program Input (from above) → **PlaylistNode.c** (Your program) → Out

Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```


©zyBooks 12/22/21 12:35 1026490
Jacob Adams
UACS100Fall2021